Introduction to Developing ACME Agents

Takaaki Shinagawa OpenVMS Security Engineering, HP

Overview

ACME (Authentication Credential Management Extension) is the new authentication subsystem provided as an EAK (Early Adopter Kit)[1] on OpenVMS Alpha Version 7.3-2. ACME provides a "plug-in" environment in which individual ACME agents for different authentication policies can be loaded independently. In addition, ACME allows application programs to perform authentication directly through the $ACM system service. Currently, HP provides the native VMS, Windows NTLM, and LDAP agents. Third parties can develop ACME agents for new authentication policies. Although the concept of ACME is very similar to PAM (Pluggable Authentication Module) on Unix platforms, ACME has a proprietary architecture and programming interfaces. Developing an ACME agent requires solid understandings of the overall ACME subsystems, interactions between ACME agents, data structures and callout and callback functions in the agent, persona extensions, and $ACM clients. The purpose of this article is to provide an overview of ACME and easy-to-understand instructions on how to develop ACME agents.

The next section, Introduction to ACME, provides an overview of the entire ACME subsystem and introduces concepts that are prerequisites for ACME agent development. Instructions for developing ACME agents and persona extensions are addressed in Section 3, Implement an ACME Agent, and Section 4, Implement a Persona Extension, respectively. Section 5, Configure ACME, shows how to configure an ACME subsystem using all of the components.

Introduction to ACME

What is an ACME Agent?

An ACME agent is a module that implements an authentication policy on an OpenVMS system. When we log into an OpenVMS system, the system asks us to enter a username and password, and then it performs authentication. By default, authentication on OpenVMS systems is performed by comparing the user-entered username and password against the SYSUAF file. This is the authentication policy implemented in the VMS ACME agent. In addition to the VMS agent, the NT and LDAP agents are available with OpenVMS Alpha today. The NT ACME agent authenticates users with the NTLM database with the Microsoft LAN Manager authentication policy, and the LDAP ACME agent authenticates users against an LDAP server.

What if we need new authentication policies for our OpenVMS operating environments? This is the motivation of developing new ACME agents. If we want new ways of authenticating users on OpenVMS systems, the answer is to develop a new ACME agent. You can develop new ACME agents for new authentication policies to accommodate your needs.

[1] Because this is an early adopter kit release, ACME should be used only for non-production purposes. The SYS$ACM component, however, has been ready for production since OpenVMS Alpha Version 7.3-1. Currently the platform for ACME agent development is OpenVMS Alpha Version 7.3-2 or higher. As of this writing (May 2004), ACME is not yet available on OpenVMS I64 (Itanium). There is no plan to provide this functionality on OpenVMS VAX.

What is the ACME subsystem?

The ACME subsystem refers to all of the components for authentication on an OpenVMS system (Figure 1). Its main feature is in its "plug-in" architecture, in which you can load individual ACME agents implementing authentication policies. An OpenVMS system administrator can load an ACME agent for a new authentication policy. For example, in order to authenticate users against an LDAP directory, the system administrator can load the LDAP ACME agent along with the VMS agent. Authentication is invoked by the $ACM system service in the application.

There are two distinct types of ACME agents: DOI and Auxiliary agents. The most significant difference is whether the agent issues credentials. A DOI agent has a capability to issue credentials, but an Auxiliary agent does not. An Auxiliary ACME agent implements specific functions that complement a DOI agent. Extra functions such as additional authentication methods (e.g. token-based and smart card), password filtering and new password checking can be implemented in Auxiliary agents.

An OpenVMS system manager can load multiple ACME agents in addition to the mandatory VMS ACME agent. When multiple DOI agents handle requests (authentication, authorization, credential generation, etc.) in an ACME subsystem, it is called a cooperative model. An independent model refers to a situation where only one DOI ACME agent processes requests and issue credentials along with the VMS ACME agent

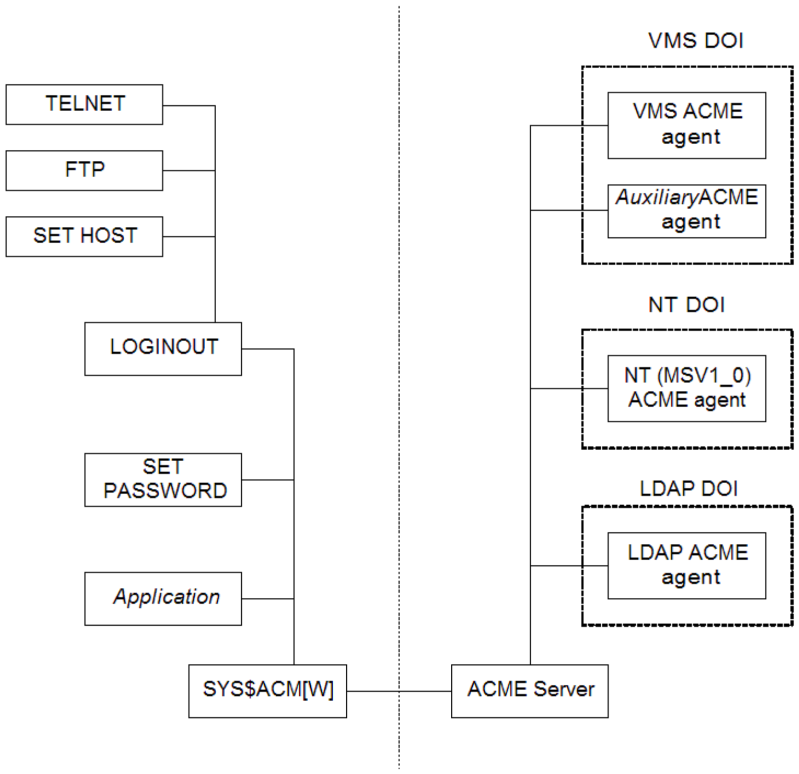


Figure 1: Overview of the ACME Subsystem

How does the ACME subsystem work?

As shown in Figure 1, the ACME subsystem is composed of the ACME server, ACME agents and the $ACM application. Applications send an authentication-related request by calling the SYS$ACM system service. The request is eventually processed by the ACME agent. In order for an OpenVMS system manager to configure the ACME subsystem, the SET SERVER ACME commands are provided on the DCL command line. A persona extension also plays an important role in conjunction with the subsystem—it is responsible for storing credentials issued by the ACME agent.

In the following lines, interactions inside the ACME subsystem are described in the order the ACME subsystem is configured and processes requests.

1. If all the ACME components have been in place, the first step to using the ACME subsystem is to start the ACME server (SET SERVER ACME/START). The ACME server can be considered as an engine of the ACME subsystem— it dispatches requests from a $ACM application to ACME agents.

2. Once the ACME server is started, it becomes possible to load ACME agents. When an ACME agent is being loaded in the ACME subsystem (SET SERVER ACME/CONFIGURE), the ACME$CO_AGENT_INITIALIZE control callout function is executed in the agents (Figure 2).

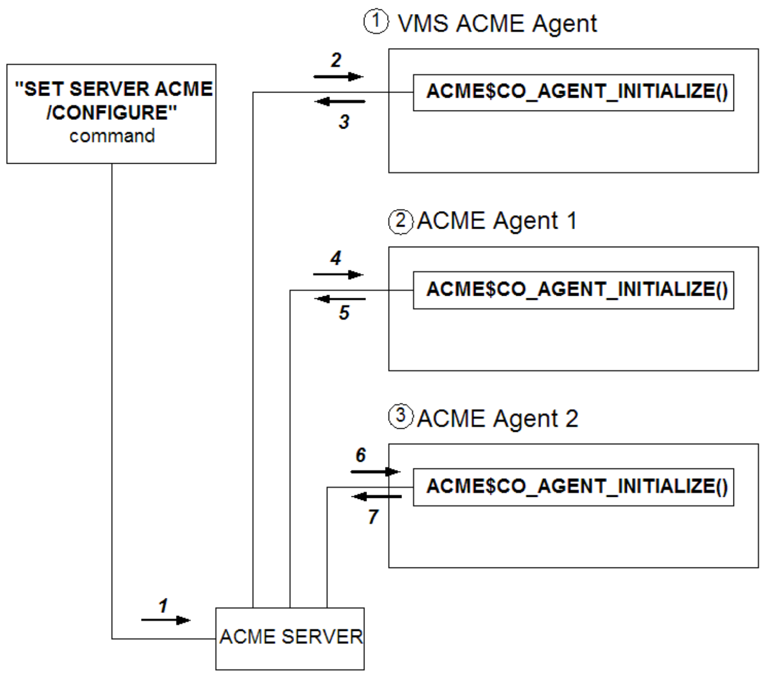

① VMS ACME Agent
"SET SERVER ACME /CONFIGURE" command
2
3
ACME$CO_AGENT_INITIALIZE()
②ACME Agent 1
4
5
ACME$CO_AGENT_INITIALIZE()
③ACME Agent 2
6
7
ACME$CO_AGENT_INITIALIZE()
1
ACME SERVER

Figure 2: ACME Control Flow for ACME$CO_AGENT_INITIALIZE

3. To activate request dispatching, the system manager runs the SET SERVER ACME /ENABLE command. At this time, the ACME$CO_AGENT_STARTUP routine in each ACME agent specified in the command is executed (Figure 3).

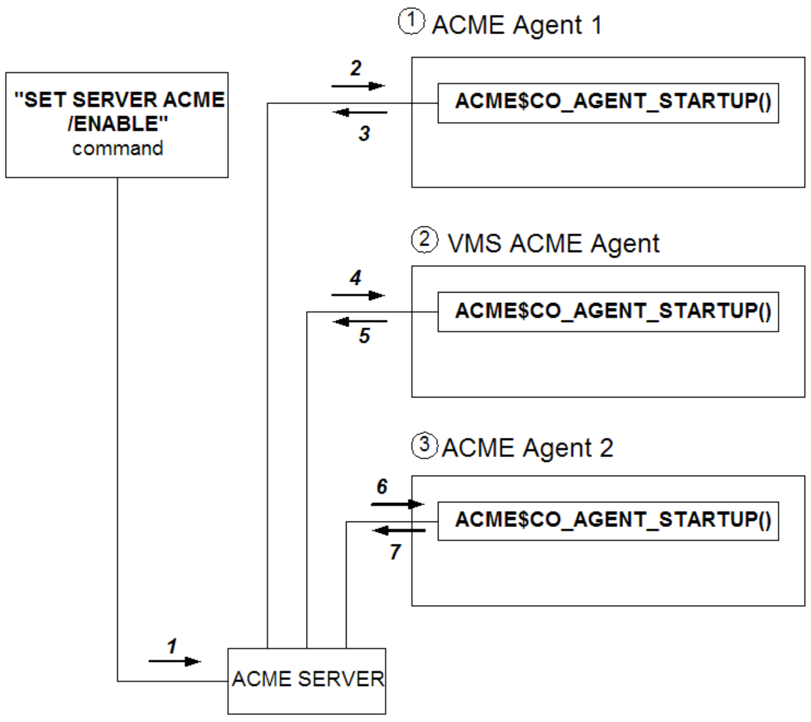


Figure 3: ACME Control Flow for ACME$CO_AGENT_STARTUP

4. Once the dispatching is enabled successfully, the ACME server and agents are ready to process the requests from the $ACM application. Every Authenticate Principal or Change Password request is performed using the following steps:

   4-1. The application calls the $ACM system service. The ACME$_FC_AUTHENTICATE_PRINCIPAL function code is specified for the Authenticate Principal request, and ACME$_FC_CHANGE_PASSWORD is specified for the Change Password request.

   4-2. The ACME server receives the request from the $ACM system service and dispatches it to the ACME agents.

   4-3. The requests are processed by the ACME agents in the order the agents are loaded.

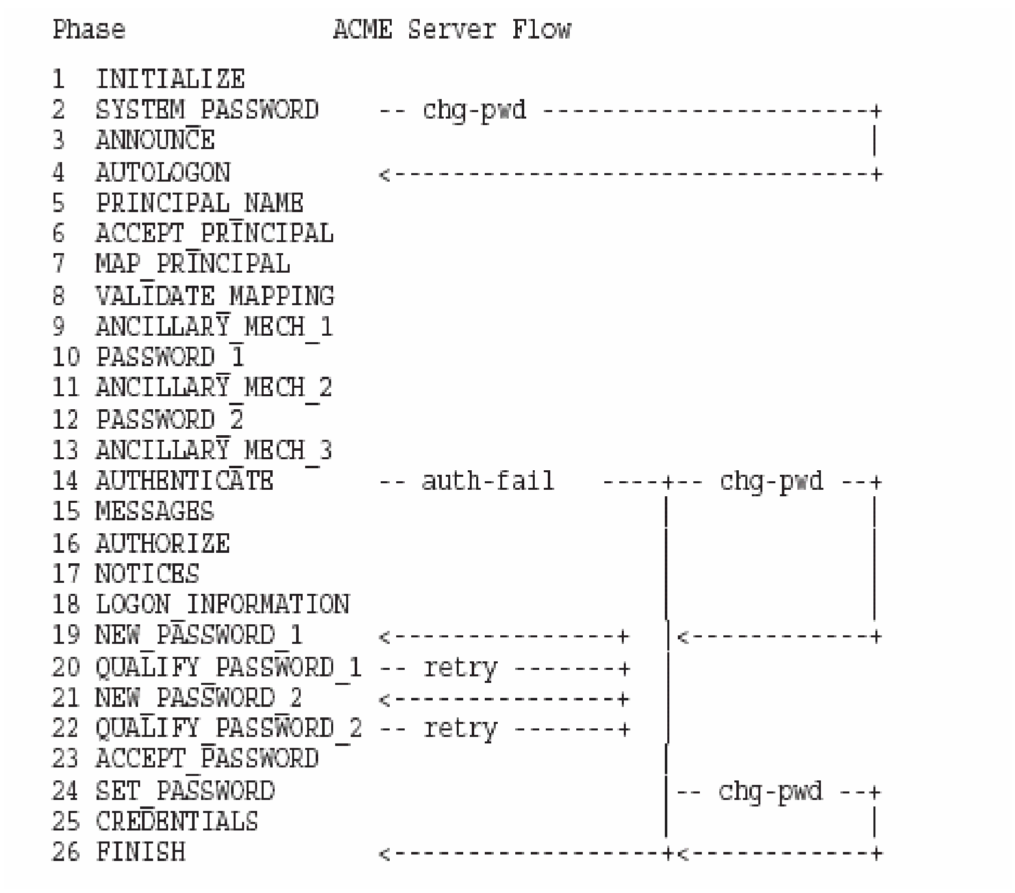
   4-4. Each time the request is processed by a callout routine, a return value from the routine to the ACME server determines the request's flow.

   - If ACME$_CONTINUE is returned, the ACME server dispatches the request to the next callout routine.
   - If the agent returns ACME$_PERFORMDIALOGUE, the ACME server performs a specified input/output dialogue, and then the request is dispatched to the same callout routine again.

- If the agent returns ACME$_FAILURE, the server dispatches the request to the final callout routine, ACME$CO_FINISH.
- If the agent returns ACME$_AUTHFAILURE, the ACME server continues processing through the ACME$CO_AUTHENTICATE callout routine, and then the service will return ACME$_AUTHFAILURE to the $ACM client.

For more return values, refer to Table 1-2 in the *ACME Developer's Guide*.

4-5. Repeat Steps 4-3 through 4-5 until you complete the ACME$CO_FINISH callout routine. Figure 4 shows all phases and request flows for authenticate-principal and change-password requests.



```
Phase              ACME Server Flow

1  INITIALIZE
2  SYSTEM_PASSWORD    -- chg-pwd ----------------------+
3  ANNOUNCE                                            |
4  AUTOLOGON          <--------------------------------+
5  PRINCIPAL_NAME
6  ACCEPT_PRINCIPAL
7  MAP_PRINCIPAL
8  VALIDATE_MAPPING
9  ANCILLARY_MECH_1
10 PASSWORD_1
11 ANCILLARY_MECH_2
12 PASSWORD_2
13 ANCILLARY_MECH_3
14 AUTHENTICATE       -- auth-fail   ----+-- chg-pwd --+
15 MESSAGES                              |             |
16 AUTHORIZE                             |             |
17 NOTICES                               |             |
18 LOGON_INFORMATION                     |             |
19 NEW_PASSWORD_1     <---------------+  |<------------+
20 QUALIFY_PASSWORD_1 -- retry -------+  |
21 NEW_PASSWORD_2     <---------------+  |
22 QUALIFY_PASSWORD_2 -- retry -------+  |
23 ACCEPT_PASSWORD                       |
24 SET_PASSWORD                          |-- chg-pwd --+
25 CREDENTIALS                           |             |
26 FINISH             <------------------+<------------+
```

Figure 4: Authentication and change-password request phases and flow

Both Authenticate Principal and Change Password requests follow the procedures described above. There are, however, some differences in callout routines that these two types of requests go through. As shown in Figure 4, the Authenticate Principal and Change Password requests are processed through different series of callout routines. Some callout routines are specific to either type of requests. Callout routines from ACME$CO_NEW_PASSWORD_1 through ACME$CO_SET_PASSWORD (Figure 4) are used to handle one or two new password(s)—these are essential to a change-password request. But an authentication request does not go though these routines unless the password is expired, and a new password must be obtained from the user. For a change-password request, the ACME server doesn't dispatch the request to the ACME$CO_ANNOUNCE, ACME$CO_MESSAGES, ACME$CO_AUTHORIZE, ACME$CO_NOTICES, ACME$CO_LOGON_INFORMATION, and ACME$CO_CREDENTIALS routines. The next section describes more details about callout routines.

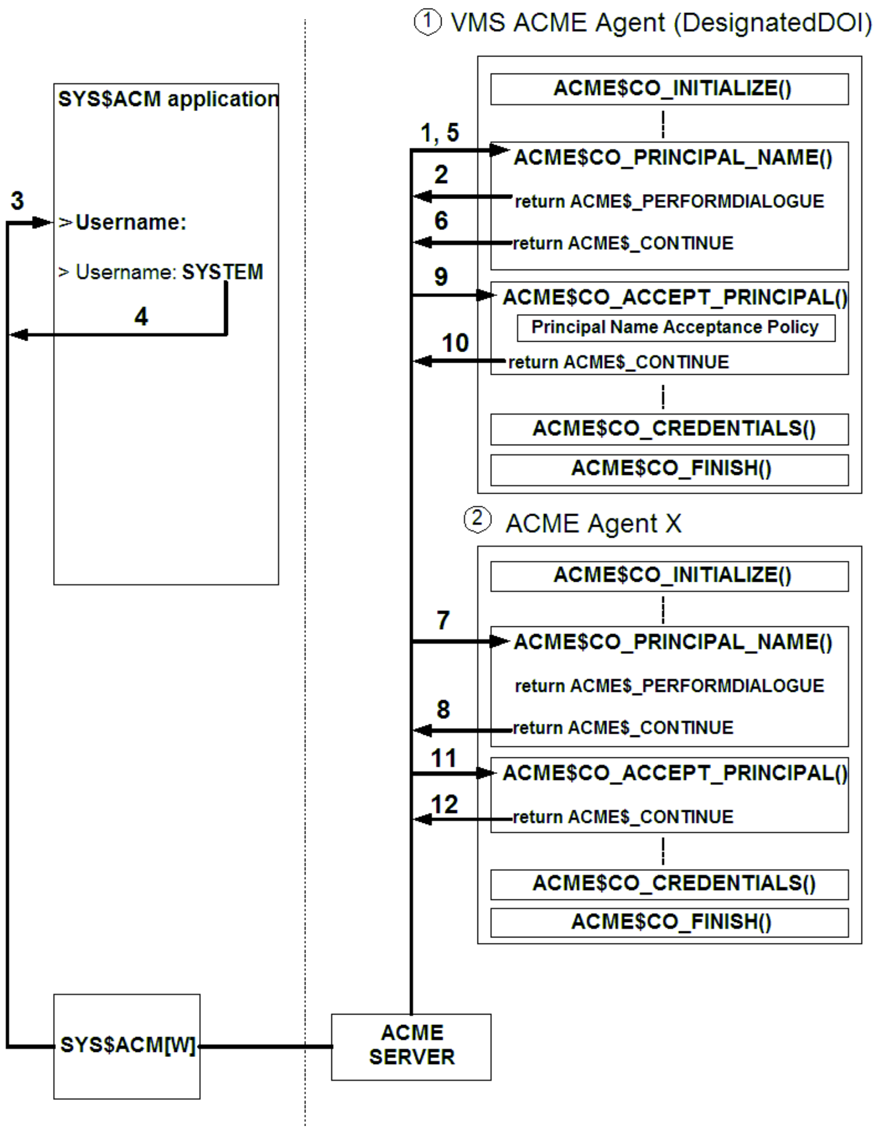


Figure 5: ACME Control Flow for the PRINCIPAL_NAME and ACCEPT_PRINCIPAL phases

Figure 5 illustrates ACME control flow in the PRINCIPAL_NAME and ACCEPT_PRINCIPAL phases. Whereas Figure 4 shows sequence and flow only in an ACME agent, Figure 5 shows the ACME control flow when multiple ACME agents are loaded. After completing the INITIALIZE through AUTO_LOGON phases, the ACME dispatches the request to the ACME$CO_PRINCIPAL_NAME callout routine of the ACME agent loaded first (1 in Figure 5). To queue a username prompt (Username:) to the $ACM client, this callout routine returns ACME$_PERFORMDIALOGUE (2). The ACME server sends the prompt string to the $ACM application (3). After the user enters a username (4), the request is dispatched again to the PRINCIPAL_NAME callout routine (5). If the principal name is received, the callout routine returns ACME$_CONTINUE (6). If only one ACME agent is loaded, the request is dispatched into the ACCEPT_PRINCIPAL routine for the next step. In this example, however, the ACME server dispatches the request to the same phase in the agent loaded in the second place (7). Because this phase has already been completed by the first agent, the second agent returns ACME$_CONTINUE (8). Then, the ACME server dispatches the request to the next phase, ACCEPT_PRINCIPAL, in the first agent (9). If the principal name policy accepts the user-entered principal name, the ACME$CO_ACCEPT_PRINCIPAL routine returns ACME$_CONTINUE (10), and the request is dispatched to the same callout routine in the second agent (11). The second agent is not a Designated DOI agent (the Designated DOI should have been declared by the first one in the ACCEPT_PRINCIPAL phase or before). Thus it simply returns ACME$_CONTINUE (12). The request will be dispatched by the ACME server in the same way through the FINISH phase.

Upon successful authentication, a $ACM client can acquire credentials from the ACME agent that supports issuing credentials in the CREDENTIALS callout routine. The ACME agent issues credentials to a persona extension associated with the $ACM application by calling the ACMEKCV$CB_ISSUE_CREDENTIALS callback function in its ACME$CO_CREDENTIALS callout routine. Once the credentials have been issued into the persona extension, the $ACM application can perform operations with the credentials by calling the PERSONA system services such as SYS$PERSONA_ASSUME and SYS$PERSONA_QUERY. To retrieve and handle the credentials, the application must support the formats of the credentials. For example, an application that supports only the VMS credentials cannot handle those newly defined in another persona extension. It is essential that the $ACM client, ACME agents, and persona extensions agree on the credentials formats. Figure 6 depicts the whole process of issuing and acquiring the credentials.

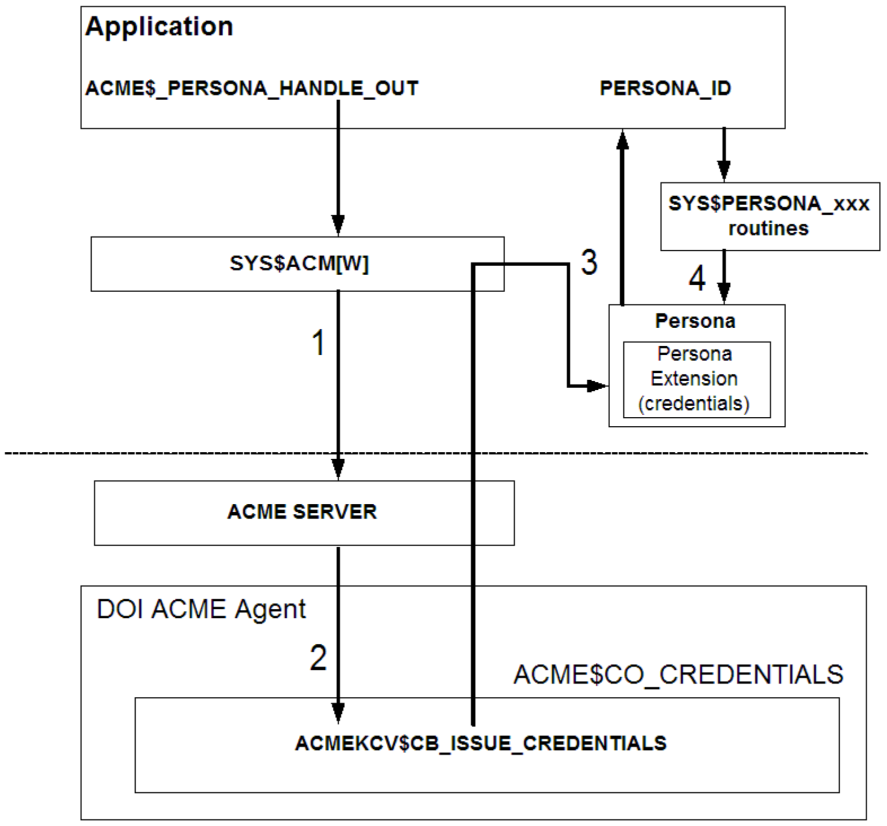


Figure 6: Credential generation process

In addition to the authenticate-principal and change-password requests, there are three types of requests from a SYS$ACM system service: QUERY, EVENT, and RELEASE_CREDENTIALS. Their function codes with the SYS$ACM are ACME$_FC_QUERY, ACME$_FC_EVENT and ACME$_FC_RELEASE_CREDENTIALS. The QUERY and EVENT requests are processed by the ACME$CO_QUERY and ACME$CO_EVENT callout routines in an ACME agent, respectively. The RELEASE_CREDENTIALS request simply deletes the credentials in the persona extension. The implementation of these callout routines will be discussed in the next section.

When a system is terminating the ACME subsystem or disabling ACME agents, ACME$CO_AGENT_SHUTDOWN is executed in the same manner as ACME$CO_AGENT_STARTUP (Figure 3).

What do we have to develop?

As mentioned above, if you are developing an ACME agent that issues credentials, we also need to develop its persona extension as well as an ACME agent. The next section presents instructions for developing an ACME agent, and the following section describes how to implement a persona extension. Also, if the credential type and input requested by the agent are not supported by the existing $ACM application, it is necessary to develop a new one or modify the application.

Implement an ACME Agent

Determine the type of the ACME agent

Before you start the implementation, decide on the type of the ACME agent. The ACME agent type is either a DOI or Auxiliary agent. As mentioned in the Introduction to ACME section, a DOI agent issues credentials. An Auxiliary ACME agent does not work alone—it is designed to provide additional functionality such as complementary authentication in conjunction with a DOI agent.

An DOI agent acts as the Designated DOI agent when it is the target of the $ACM call. If you are developing a DOI agent, you also need to decide whether it can function as a Secondary DOI agent in the cooperative model for an untargeted $ACM call. In the cooperative model, each of multiple DOI agents (both Designated and Secondary DOI agents) is expected to be responsible for authentication and issuing credentials. In the independent model, only the DOI agent issues credentials—Secondary DOI agents other than the VMS agent do not perform those operations.

It is desirable to know all other ACME agents with which your agent will be configured in the ACME subsystem. When loading multiple ACME agents, the collection of those rules in all agents defines the operation model (either cooperative or independent model). The ACME server or ACME subsystem doesn't control behaviors of ACME agents to conform the specific operation model. Instead, rules implemented in each agent and occasionally order of agents make differences in their operations. The more details you know about other agents, the easier it becomes to develop an ACME agent that works correctly with those agents.

Start implementing an ACME agent

Once you decide the ACME agent type, it is time to start implementing the ACME agent. The example below is based on the example code shipped in OpenVMS Alpha (SYS$EXAMPLES:ACME_EXAMPLE_DOI_ACME.C). This example agent is a DOI agent for the independent model and demonstrates the basic implementation necessary for any agent. If you are new to ACME agent development, it is highly recommended that you create a new one based on the example ACME agent.

Define data structures

In the ACME agent, you define three data structures to store data specific to the agent or each request. The ACME subsystem uses ACME data structures such as WQE (Work Queue Entry) and KCV (Kernel Callback Vector), but they are already predefined in the system header files such as acmedef.h.

The acme_context data structure

Information in the acme_context data structure is kept as long as the ACME agent is active (from startup until shutdown). Thus, data such as authentication success/failure and the number of requests can be defined and stored in this data structure.

The request_context (= wqe_context) data structure

The request_context data structure is also called wqe_context— they are synonyms. Information in the request_context data structure is kept during a single request. Authentication data and user/account specific information associated with an authentication request can be defined and stored in this data structure. For example, a username and password received from the $ACM client are stored.

The credential data structure

If the ACME agent issues credentials, this data structure is essential. All credential fields must be defined in a credential data structure. The definitions of the credential data must be the same as that in the persona extension source code.

Implement callout routines

The major task of developing an ACME agent is to implement a series of callout routines. They can be classified into the following three types:

- control callout routines
- authenticate-principal/change-password callout routines
- event and query callout routines

The steps that should be implemented in those callout routines are described below. For implementation in C, refer to the example program (ACME_EXAMPLE_DOI_ACME.C) in SYS$EXAMPLES.

This article provides comprehensive procedures for callout functions. For more details such data types of arguments and item/function codes, refer to the *ACME Developer's Guide*.

Control callout routines

The control routines are ACME$CO_AGENT_INITIALIZE, ACME$CO_AGENT_STARTUP, ACME$CO_AGENT_STANDBY, and ACME$CO_AGENT_SHUTDOWN. As explained in the Introduction to ACME section, these control callout routines are invoked when a system manager loads, starts, suspends, and shuts down the ACME agents/server.

ACME$CO_AGENT_INITIALIZE

This routine is executed when loading the ACME agent (with SET SERVER ACME /CONFIGURE command).

1. Verify the revision levels of the WQE and KCV data structures. The purpose is to check the compatibility of data structure versions of the ACME server and the ACME agent. A structure revision level should be checked in WQE, whereas both ACM kernel and structure revision levels are checked in KCV. In each revision levels, major and minor versions exist. The major versions need to be equal, and the minor version from the ACME server (in WQE) needs to be greater than or equal to the one with the ACME agent. If the

revision levels mismatch, return ACME$_UNSUPREVLVL. For more details in the revision levels, refer to Section 4.1.1 and 4.1.2 in the *ACME Developer's Guide*.

2. Initialize ACME Resource Block (ACMERSRC). Although this initialization is optional for most fields in ACMERSRC, it is required to explicitly set some fields such as privileges depending on the ACME agent's operations. More information about ACME Agent Resource Requirements Block is available in Section B.5 in the *ACME Developer's Guide*.
3. Set the ACME agent's name and report it to the ACME server with ACMEKCV$CB_REPORT_ATTRIBUTES().
4. Set the current status string and report it to the ACME server with ACMEKCV$CB_REPORT_ACTIVITY().
5. Return ACME$_CONTINUE.

ACME$CO_AGENT_STARTUP

This routine is executed when activating the ACME agent (with the SET SERVER ACME /ENABLE command).

1. Allocate the acme_context data structure with ACMEKCV$CB_ALLOCATE_ACME_VM().
2. Initialize the counters in acme_context if they exist.
3. Return ACME$_CONTINUE.

ACME$CO_AGENT_SHUTDOWN

This routine is executed when stopping (with SET SERVER ACME /EXIT) or disabling (with SET SERVER ACME /DISABLE) the ACME agent.

1. Deallocate the acme_context data structure with ACMEKCV$CB_DEALLOCATE_ACME_VM().
2. Return ACME$_NORMAL upon success.

ACME$CO_AGENT_STANDBY

This routine is executed when temporarily disabling (with SET SERVER ACME /SUSPEND) the ACME agent. The purpose of this operation is to close files for a possible system backup operation. Note that there is no corresponding resume callout routine. The request processing resumes when the operator issues the SET SERVER ACME/RESUME command. The example agent does not implement any operation in this callout routine and returns ACME$_CONTINUE.

Request callout functions

The request routines are invoked by an authentication request from a $ACM client.

ACME$CO_INITIALIZE

This is the first routine every authenticate-principal/change-password request goes through. ACME$CO_INITIALIZE allocates request_context, which is the context data structure for a request, and process common and ACME-specific item lists. If the request is in the dialogue mode, those lists are empty. In the non-dialogue mode, however, items such as a username and password can be obtained in this routine.

1. If another ACME agent is targeted by the $ACM client, return ACME$_CONTINUE (skip the rest of the steps in this ACME agents). This check is done by comparing ACME numbers of ACMEWQE$L_TARGET_ACME_ID and ACMEWQE$L_CURRENT_ACME_ID.
2. If another ACME agent has already declared as a Designated DOI agent, return ACME$_CONTINUE. This check is done by comparing ACME numbers of ACMEWQE$L_DESIGNATED_ACME_ID and ACMEWQE$L_CURRENT_ACME_ID.
3. Allocate and initialize the request-specific data structure (request_context).
4. Set the target status (whether the agent is targeted or not), and update target status value in request_context.
5. Process the common item list (principal name and password may be supplied by the $ACM client).
6. Process the ACME specific item list (ACME-specific items may be supplied by the $ACM client).
7. Return ACME$_CONTINUE.

ACME$CO_SYSTEM_PASSWORD

This callout routine is intended to be implemented only in the VMS agent. Other agents simply return ACME$_CONTINUE. In this routine, the VMS agent obtains a system password for authentication. This phase is associated with the ACME$_PASSWORD_SYSTEM item code with the $ACM system service.

ACME$CO_ANNOUNCE

This routine is invoked by the ACME server only for the Authenticate Principal request. It is implemented to display information to the user prior to the username prompt. The VMS agent displays the message defined with the SYS$ANNOUNCE logical. The example agent doesn't output any message by simply returning ACME$_CONTINUE. To display a message, however, implement the following procedure.

1. If this is the first time to enter this routine:
   - The agent sends the message to the $ACM client through ACMEKCV$CB_QUEUE_DIALOGUE().
   - Return ACME$_PERFORMDIALOGUE.
2. If this is the second time to enter this routine:
   - Return ACME$_CONTINUE.

ACME$CO_AUTOLOGON

If the ACME agent determines a principal name automatically (i.e. without user intervention), the mechanism is implemented in this callout routine. If this capability is necessary, follow the steps below.

1. Check the ACMEWQEFLG$V_PHASE_DONE flag. If this phase has been completed by another agent, return ACME$_CONTINUE.
2. Implement the agent's specific mechanism to determine a principal name.
3. Load the principal name string and length in WQE through ACMEKCV$CB_SET_WQE_PARAMETER.
4. Set the "phase done" flag (ACMEWQEFLG$K_PHASE_DONE) in WQE through ACMEKCV$CB_SET_WQE_FLAG().
5. Return ACME$_CONTINUE.

ACME$CO_PRINCIPAL_NAME

In the dialogue mode, this routine is responsible for prompting and obtaining a principal name string from the $ACM client.

1. Check the ACMEWQEFLG$V_PHASE_DONE flag. If this phase has been completed by another agent, return ACME$_CONTINUE.
2. If another ACME agent has already declared as a Designated DOI agent, return ACME$_CONTINUE (skip the rest of the steps in this ACME agents).
3. If this is the first time to enter this routine:
   - If a principal name has already been provided, return ACME$_CONTINUE.
   - If the principal name is not in the buffer:
     - Queue the principal name prompt to the $ACM client by calling ACMEKCV$CB_QUEUE_DIALOGUE().
     - Set a flag that indicates the request has already entered this routine once. Since returning ACME$_PERFORMDIALOGUE causes the request to be returned to this routine as soon as completing this dialogue, this flag is necessary to know that the next time the request enters this routine is the second time.
     - Return ACME$_PERFORMDIALOGUE.
4. If this is the second time to enter this routine:
   - If the ACME$_PRINCIPAL_NAME_IN item code is not in the common item list, return ACME$_FAILURE.
   - Load the principal name string and length in WQE through ACMEKCV$CB_SET_WQE_PARAMETER.
   - Set the "phase done" flag (ACMEWQEFLG$K_PHASE_DONE) in WQE through ACMEKCV$CB_SET_WQE_FLAG().
   - Return ACME$_CONTINUE.

ACME$CO_ACCEPT_PRINCIPAL

This callout routine examines a principal obtained in the ACME$_CO_PRINCIPAL_NAME routine. If the principal name is accepted by the principal name policy, this routine must declare as a Designated DOI agent. This is the last phase in which an ACME agent can set the Designated DOI status.

1. Check the ACMEWQEFLG$V_PHASE_DONE flag. If this phase has been completed by another agent, return ACME$_CONTINUE.
2. If another ACME agent has already declared as a Designated DOI agent, return ACME$_CONTINUE (skip the rest of the steps in this ACME agents).
3. Load the principal name and length in request_context from WQE. This is necessary when the ACME$CO_ACCEPT_PRINCIPAL routine was completed by another agent. In this situation, the principal and its length fields in the request_context is empty, whereas they are stored in the WQE.
4. Implement a policy to accept a principal name:
    - If the principal is not accepted by the policy, return ACME$_AUTHFAILURE.
5. Set the accepted principal name in WQE thorough ACMEKCV$CB_SET_WQE_PARAMETER().
6. Declare this is the Designated DOI agent.
7. Set the "phase done" flag (ACMEWQEFLG$K_PHASE_DONE) in WQE through ACMEKCV$CB_SET_WQE_FLAG().
8. Return ACME$_CONTINUE.

ACME$CO_MAP_PRINCIPAL

This routine specifies the VMS username in the SYSUAF file which corresponds to the principal name accepted in the previous phases. If the accepted principal name is identical to the VMS username, no mapping is necessary. Because VMS usernames are uppercased, uppercasing is required in the mapping process. For example, an accepted principal name john@openvms may be mapped to JOHN in the UAF record in this routine.

1. Check the ACMEWQEFLG$V_PHASE_DONE flag. If this phase has been completed by another agent, return ACME$_CONTINUE.
2. If this agent is not participating in this request (address of request_context is null), return ACME$_CONTINUE.
3. If this agent is not the Designated DOI agent, return ACME$_CONTINUE.
4. Implement the mapping policy.
5. Convert the principal string to uppercase. This conversion is necessary because the current ACME server doesn't uppercase the principal string. Without uppercasing, a lowercase/mixed-case principal string causes a mismatch against a VMS username, which is always uppercased.
6. Set the uppercased principal in WQE through ACMEKCV$CB_SET_WQE_PARAMETER().
7. Return ACME$_CONTINUE.

ACME$CO_VALIDATE_MAPPING

Optionally, an ACME agent can check the validity of the mapped username in this routine. The VMS agent always checks the mapped VMS username in this phase. If the mapped username doesn't exist in the SYSUAF file, the request is terminated by the VMS agent.

ACME$CO_ANCILLARY_MECH_1

This routine is used when the ACME agent collects additional information from the $ACM client. The example ACME agent doesn't implement this routine because no information other than a username and password is used. However, if an ACME agent uses other types of user credentials such as a cryptographic token, it is expected to obtain them in this callout routine or the other two ancillary mechanism routines (ACME$CO_ANCILLARY_MECH_2 and ACME$CO_ANCILLARY_MECH3).

ACME$CO_PASSWORD_1

In the dialogue mode, this routine is responsible for prompting and obtaining a password from the $ACM client.

1. Check the ACMEWQEFLG$V_PHASE_DONE flag. If this agent is not participating in this request or this phase has been completed by another agent (address of request_context is null), return ACME$_CONTINUE.
2. If the pre-authenticated flag (ACMEWQEFLG$V_PREAUTHENTICATED) has been set:
   - Return ACME$_AUTHFAILURE, or if the agent allows the pre-authenticated mode, return ACME$_CONTINUE.
3. If another ACME agent has already declared as a Designated DOI agent, return ACME$_CONTINUE (skip the rest of the steps in this ACME agents).
4. If this is the first time to enter this routine:
   - If a password has already been provided, return ACME$_CONTINUE.
   - If the password is not in the buffer:
     - The password prompt will be sent to the $ACM client through ACMEKCV$CB_QUEUE_DIALOGUE().
     - Set a flag that indicates the request has already entered this routine once. Since returning ACME$_PERFORMDIALOGUE causes the request to be returned to this routine as soon as completing this dialogue, this flag is necessary to know that the next time the request enters this routine is the second time.
     - Return ACME$_PERFORMDIALOGUE.
5. If this is the second time to enter this routine:
   - If the ACME$_PASSWORD_1 item code is not in the common item list, return ACME$_FAILURE.
   - Load the password string and length in WQE through ACMEKCV$CB_SET_WQE_PARAMETER.
   - Set the "phase done" flag (ACMEWQEFLG$K_PHASE_DONE) in WQE through ACMEKCV$CB_SET_WQE_FLAG().
   - Return ACME$_CONTINUE.

ACME$CO_ANCILLARY_MECH_2

This routine is used when the ACME agent collects any ACME-specific authentication information between the ACME$_PASSWORD_1 and ACME$_PASSWORD_2 phases. The example ACME agent doesn't implement this routine.

ACME$CO_PASSWORD_2

If the ACME agent needs a second password for authentication, this routine should be implemented. The implementation should follow the same procedures as for the ACME$CO_PASSWORD_1 routine.

ACME$CO_ANCILLARY_MECH_3

This routine is used when the ACME agent collects any ACME-specific authentication information between the ACME$_PASSWORD_1 and ACME$_PASSWORD_2 phases. The example ACME agent doesn't implement this routine.

ACME$CO_AUTHENTICATE

An ACME agent-specific authentication policy is implemented in this routine.

1. If this agent is not participating in this request (address of request_context is null), return ACME$_CONTINUE.
2. If this is not a Designated DOI agent, return ACME$_CONTINUE.
3. If the pre-authenticated flag (ACMEWQEFLG$V_PREAUTHENTICATED) has been set, return ACME$_AUTHFAILURE.
4. If the agent allows the pre-authenticated mode, return ACME$_CONTINUE.
5. Implement an authentication policy. The example agent implements a very simple authentication policy. It simply checks the first character of the password. If the password starts with 'a', the request is authenticated. Otherwise, authentication fails. In ACME agents for production use, it would be necessary to communicate with a file or database storing user credentials such as passwords.
   - If authentication succeeds, return ACME$_CONTINUE.
   - If authentication fails, return ACME$_AUTHFAILURE.

ACME$CO_MESSAGES

This routine is invoked by the ACME server only for the Authenticate Principal request. This is used to output any text message to the $ACM client between the ACME$CO_AUTHENTICATE and ACME$CO_AUTHORIZE phases. The implementation will be similar to other messaging callout routines such as ACME$CO_ANNOUNCE.

1. If this is the first time to enter this routine:
   - The message will be sent to the $ACM client through ACMEKCV$CB_QUEUE_DIALOGUE().
   - Return ACME$_PERFORMDIALOGUE.
2. If this is the second time to enter this routine, return ACME$_CONTINUE.

ACME$CO_AUTHORIZE

This routine is invoked by the ACME server only for the Authenticate Principal request. It performs authorization for the authenticated request. The request is authorized based on the ACME agent-specific authorization policy. Access restrictions such as privileges, modes of operation, and account duration are examined to grant the authorization. If there is no authorization policy with the ACME agent, this callout routine is optional.

ACME$CO_NOTICES

This routine is invoked by the ACME server only for the Authenticate Principal request. This is used to output a long text message to the $ACM client after successfully completing the authentication and authorization phases. For example, the VMS agent displays a message defined with SYS$WELCOME. The implementation will be similar to other messaging callout routines such as ACME$CO_ANNOUNCE and ACME$CO_MESSAGES.

1. If this is the first time to enter this routine:
   - The message will be sent to the $ACM client through ACMEKCV$CB_QUEUE_DIALOGUE().
   - Return ACME$_PERFORMDIALOGUE.
2. If this is the second time to enter this routine:
   - Return ACME$_CONTINUE.

ACME$CO_LOGON_INFORMATION

This routine is invoked by the ACME server only for the Authenticate Principal request. This is used to output text information to the $ACM client after the authorization phase. The difference from the ACME$CO_NOTICES callout routine is that this routine displays short, critical logon information. This will be the final output text after authentication and authorization. The VMS agent uses this phase to display last logon time and number of logon failures. The implementation will be similar to other messaging callout routines such as ACME$CO_ANNOUNCE and ACME$CO_MESSAGES.

1. If this agent is not participating in this request (address of request_context is null), return ACME$_CONTINUE.
2. If this is not a Designated DOI agent, return ACME$_CONTINUE.
3. If this is the first time to enter this routine:
   - If dialogue is not possible, skip the rest of this routine (return ACME$_CONTINUE).
   - Send a logon message to the $ACM client through ACMEKCV$CB_QUEUE_DIALOGUE().
   - Return ACME$_PERFORM DIALOGUE.
4. If this is the second time to enter this routine, return ACME$_CONTINUE.

ACME$CO_NEW_PASSWORD_1

This callout routine is mainly for change-password requests. The only situation in which this routine is used for authenticate-principal requests is when the password has expired. It prompts and obtain a new password for the $ACM client. The implementation is similar to the ACME$CO_PASSWORD_1 routine.

1. Check the address of request_context and ACMEWQEFLG$V_PHASE_DONE flag. If this agent is not participating in this request or this phase has been completed by another agent, return ACME$_CONTINUE.
2. If the ACMEWQEFLG$V_SKIP_NEW_PASSWORD flag is set, return ACME$_CONTINUE.
3. If this is not a Designated DOI agent, return ACME$_CONTINUE.
4. If the request in the AUTHENTICATE_PRINCIPAL mode and the PasswordExpired flag is not set, return ACME$_CONTINUE. As mentioned above, unless a password is expired, an authenticate-principal doesn't need this phase.
5. If this is the first time to enter this routine:
    - If a new password has already been provided, return ACME$_CONTINUE.
    - If the new password is not in the buffer:
        - If either input or noecho can't be performed, return ACME$_INSFDIALSUPPORT.
        - The new password prompt will be sent to the $ACM client through ACMEKCV$CB_QUEUE_DIALOGUE().
        - Set a flag that indicates the request has already entered this routine once. Since returning ACME$_PERFORMDIALOGUE causes the request to be returned to this routine as soon as completing this dialogue, this flag is necessary to know that the next time the request enters this routine is the second time.
        - Return ACME$_PERFORMDIALOGUE.
6. If this is the second time to enter this routine:
    - If the ACME$_NEW_PASSWORD_1 item code is not in the common item list, return ACME$_FAILURE.
    - Load the new password string and length in WQE through ACMEKCV$CB_SET_WQE_PARAMETER.
    - Set the "phase done" flag (ACMEWQEFLG$K_PHASE_DONE) in WQE through ACMEKCV$CB_SET_WQE_FLAG().
    - Return ACME$_CONTINUE.

ACME$CO_QUALIFY_PASSWORD_1

This callout routine implements a policy to accept a new password. The length and special/numerical characters in a new password are usually examined.

1. Check the address of wqe_context and ACMEWQEFLG$V_PHASE_DONE flag. If this agent is not participating in this request or this phase has been completed by another agent, return ACME$_CONTINUE.
2. If the ACMEWQEFLG$V_SKIP_NEW_PASSWORD flag is set, return ACME$_CONTINUE.

3. If this is the first time to enter this routine:
    - If this is not a Designated DOI agent, return ACME$_CONTINUE.
    - If the request in the AUTHENTICATE_PRINCIPAL mode and the PasswordExpired flag is not set, return ACME$_CONTINUE.
    - If the new password is not in the buffer, return ACME$_FAILURE.
    - If the new password is in the buffer:
        - Implement a policy to accept or reject a new password.
            - If the new password is accepted:
                - Update the new password in the password database/file.
                - Set the "phase done" flag (ACMEWQEFLG$K_PHASE_DONE) in WQE through ACMEKCV$CB_SET_WQE_FLAG().
                - return ACME$_CONTINUE.
            - If the new password is rejected:
                - A message indicating that the new password was rejected is sent to the $ACM client through ACMEKCV$CB_QUEUE_DIALOGUE().
                - Return ACME$_PERFORMDIALOGUE.
4. If this is the second time to enter this routine (only when the password was rejected):
    - Set the "phase done" flag (ACMEWQEFLG$K_PHASE_DONE) in WQE through ACMEKCV$CB_SET_WQE_FLAG().
    - To terminate the request, return ACME$_FAILURE. However, in most production systems, it is a common practice to ask the user to enter another new password. To implement this way, use ACME$_RETRYPWD. This return value causes the request to re-enter the previous routine, ACME$CO_NEW_PASSWORD_1.

ACME$CO_NEW_PASSWORD_2

If the ACME agent needs a second password for authentication, this routine should be implemented. The implementation should follow the same procedures as for the ACME$CO_NEW_PASSWORD_1 routine.

ACME$CO_QUALIFY_PASSWORD_2

If the ACME agent needs a second password for authentication, this routine should be implemented. The implementation should follow the same procedures as for the ACME$CO_QUALIFY_PASSWORD_1 routine.

ACME$CO_ACCEPT_PASSWORDS

The purpose of this routine is to prepare for the next routine. It checks the availability of the password database and other resources (e.g. network connections to the database) for the operation in the ACME$CO_SET_PASSWORD callout routine. This is to minimize the possibility of failure in the next phase.

ACME$CO_SET_PASSWORD

The role of this routine is to update the password file/database for the ACME agent. Because the example ACME agent doesn't have its password database, it is not implemented. If the password update fails in this phase, this may cause discrepancies of saved passwords in password databases of multiple ACME agents. For example, if ACME agent B fails to update a password after ACME agent A has successfully updated the password in its password file, the passwords saved for agent A and B are inconsistent. To avoid or minimize such a situation, it is recommended to check the availability of the password database/file in the previous routine, ACME$CO_ACCEPT_PASSWORDS.

ACME$CO_CREDENTIALS

This routine is invoked by the ACME server only for the Authenticate Principal request. This must be implemented in any DOI agent, which always issues credentials to the $ACM client. Only the authenticate-principal requests invoke this routine. Since credentials are passed to and stored in the persona extension image, the definition of the credential data structure must be consistent with the one in the persona extension code.

1. Check the address of request_context. If this agent is not participating in this request, return ACME$_CONTINUE.
2. If this is not a Designated DOI agent, return ACME$_CONTINUE.
3. If the credential is not requested by the $ACM client, return ACME$_CONTINUE (skip the rest).
4. Send the credential to the $ACM client through ACMEKCV$CB_ISSUE_CREDENTIALS().
5. Return ACME$_CONTINUE.

ACME$CO_FINISH

This is the last routine in the ACME agent. Cleanup operations such as memory deallocation and file I/O closure are implemented in this routine.

1. If this agent is not participating in this request (address of request_context is null), return ACME$_NORMAL.
2. Deallocate request_context.
3. Return ACME$_NORMAL.

Event and Query callout routines

The optional routines ACME$CO_EVENT and ACME$CO_QUERY are not executed in the course of the dialogue request processing. They are invoked when an application calls the $ACM system service with function codes specifying those operations. Unlike the request callout functions, these routines are always executed in the targeted mode. The $ACM client targets a specific ACME agent with the ACME$CO_EVENT or ACME$CO_QUERY call, so the ACME server calls no other agents for this callout routine. Therefore, you don’t have to consider configurations and scenarios with other ACME agents. This is the reason no specific and required internal steps exist for these routines. In both routines, item codes from the $ACM client can be found in the item list, and a specific event or information query should be implemented for each supported item code.

ACME$CO_EVENT

This routine, ACME$CO_EVENT, is implemented for event-related operations of an ACME agent. The ACME$_FC_EVENT function code with SYS$ACM in the application invokes this callout routine. Although implementing this routine is optional, it is appropriate to handle discrete events such as auditing and error checking. In this routine, the following input and output item codes are processed.

- ACME$_EVENT_DATA_IN

  The ACME$_EVENT_DATA_IN item code is an input item code. It specifies the buffer containing information applicable to an event operation. The meaning of this data is specific to the domain of interpretation for which it is used.

- ACME$_EVENT_TYPE

  The ACME$_EVENT_TYPE item code is an input item code. It specifies the type of event being reported. The buffer must contain a longword value. Interpretation of the value is specific to the domain of interpretation to which the event is being reported.

- ACME$_SERVER_NAME_IN

  Specifies the Event Server to which an Event should be directed. The meaning of this item is specific to the target domain of interpretation.

- ACME$_EVENT_DATA_OUT

  The ACME$_EVENT_DATA_OUT item code is an output item code. It specifies the buffer to receive information returned from an event operation. The meaning of this data is specific to the domain of interpretation for which it is used.

- ACME$_SERVER_NAME_OUT

  Reports the Event Server to which an Event was directed. The meaning of this item is specific to the target domain of interpretation.

ACME$CO_QUERY

This routine is used to provide the ACME agent's information with a $ACM client for function code ACME$_FC_QUERY. In this routine, it is necessary to parse values in the three input item codes (ACME$_QUERY_TYPE, ACME$_QUERY_KEY_TYPE, and ACME$_QUERY_KEY_VALUE) and return data with the output item code, ACME$_QUERY_DATA.

- ACME$_QUERY_TYPE

  The ACME$_QUERY_TYPE item code is an input item code. It specifies the type of data to be returned in the buffer described by the corresponding ACME$_QUERY_DATA item code. The ACME$_QUERY_TYPE item code requires that an ACME$_QUERY_DATA item code immediately follow it in the item list.

- ACME$_QUERY_DATA

  The ACME$_QUERY_DATA item code is an output item code. It specifies the buffer to receive the data returned from the query operation relating to the corresponding ACME$_QUERY_TYPE item code. The ACME$_QUERY_DATA item code requires that an ACME$_QUERY_TYPE item code immediately precede it in the item list.

- ACME$_QUERY_KEY_TYPE

  The ACME$_QUERY_KEY_TYPE item code is an input item code. It specifies the key type for establishing the context of a query operation. The key format is specific to the ACME agent to which the call is directed. An ACME$_QUERY_KEY_TYPE item requires that an ACME$_QUERY_KEY_VALUE item immediately follow it in the item list.

- ACME$_QUERY_KEY_VALUE

  The ACME$_QUERY_KEY_VALUE item code is an input item code. It specifies the key data for establishing the context of a query operation. An ACME$_QUERY_KEY_VALUE item requires that an ACME$_QUERY_KEY_TYPE item immediately precede it in the item list.

Implement a Persona Extension

The persona extension's role with the ACME subsystem is to store authentication credentials issued by its corresponding ACME agent. As mentioned earlier, an ACME agent issuing credentials requires its corresponding persona extension (for the mechanism of issuing credentials, refer to Figure 6). If the ACME agent doesn't issue any credentials, it is not necessary to develop a persona extension for the ACME agent. If the ACMEKCV$CB_ISSUE_CREDENTIALS callback function is called in the ACME$CO_CREDENTIALS routine, its persona extension must be installed with the ACME agent. This section provides introductory guidelines for development of a persona extension image for ACME agent developers.

Introduction to a persona and persona extension

A persona is a group of kernel-based, protected data structures storing a user’s security profile for a process/thread. Every process in the system has at least one persona, called the natural persona. The natural persona is created during process creation. The primary data structure of a persona is the Persona Security Block (PSB). The PSB contains the security profile, including UIC, system rights chains, privileges, account name, user name, auditing flags, and counters. As shown in Appendix C in the *ACME Developer's Guide*, a persona is composed of several data structures:

- Persona Security Block (PSB)
- Persona Extension Block Array (PXB_ARRAY)
- Persona Extension Block (PXB)
- Persona Extension Creation Flags (PXB_Flags)
- Persona Extension Dispatch Vector (PXDV)
- Persona Extension Registration Block (PXRB)
- Persona Extension Create Flags (Create_Flags)
- Persona Delegation Block (DELBK)
- PSB Ring Buffer (PSBRB)
- Persona Security Block Array (PSB_ARRAY)

A persona extension is a data structure called Persona Extension Block (PXB). Figure 7 illustrates how a PXB is linked to the PSB data structure. All persona extensions except the VMS extension are indexed in the PXB_ARRAY structure. Each extension is a PXB data structure that a system manager can load the agent-specific persona extension as an executive image. The Configure ACME section addresses the installation of a persona extension, and Appendix B provides DCL and SYSMAN commands for building and installing a persona extension image. In the following lines in this section, we will focus on development of a persona extension image.

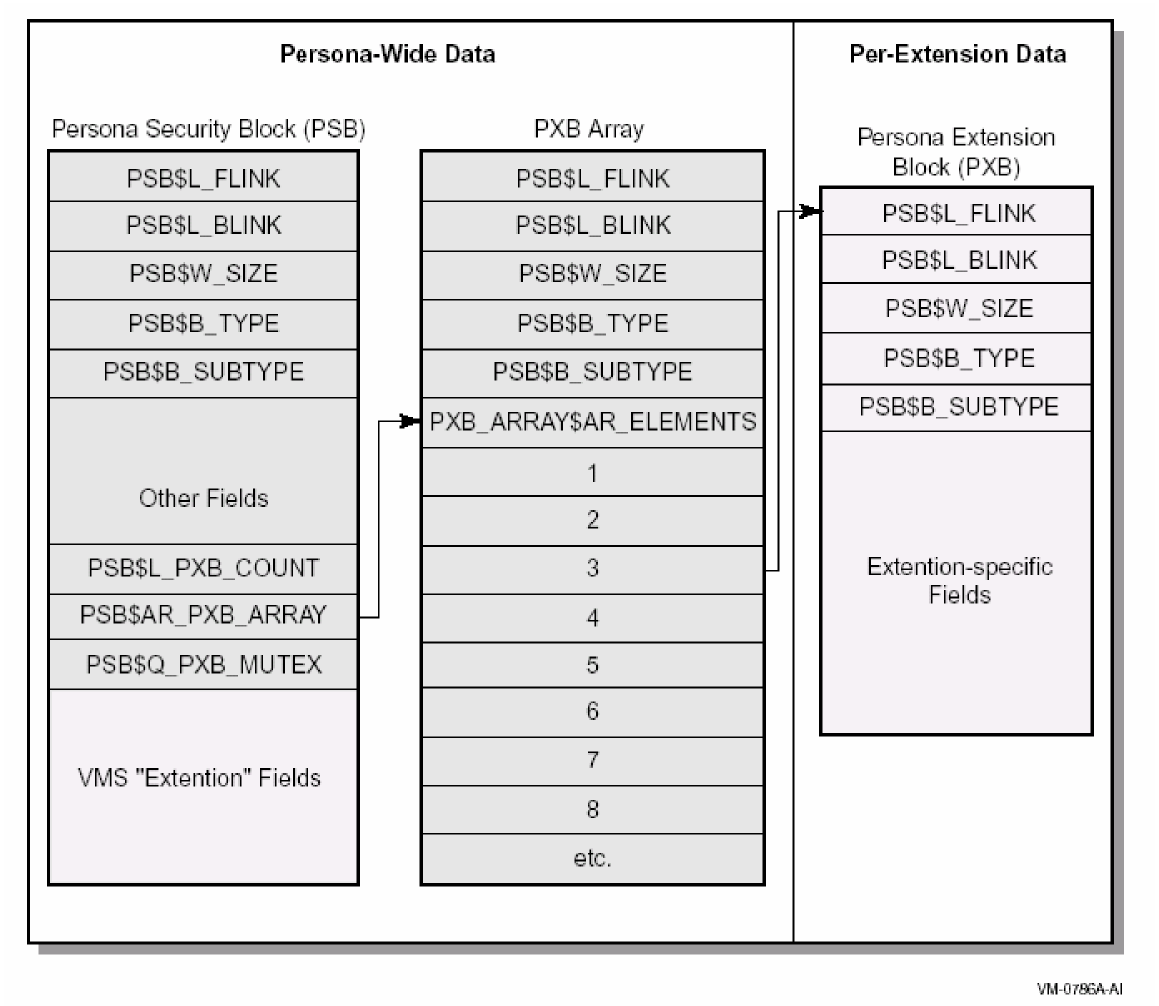


Figure 7: Some persona data structures (credentials are stored in "Extension-specific Fields")

Start implementing a person extension

The developer must follow the programming interface for a persona extension—it is different from the ACM agent's interface. The primary task of persona extension development is to implement persona extension routines. The instructions for those routines will be provided in this section.

The example persona extension code (ACME_PERSONA_EXT.C) can be found in the SYS$EXAMPLES directory. It is strongly recommended that you refer to the file while reading instructions in this section. For building and installing the example persona extension program, refer to Appendix B.

Define data structures in the persona extension program

As listed above, the entire persona extension system uses several data structures. In the persona extension program, only the credential and Persona Extension Block (PXB) data structures must be defined — both are specific to the persona extension.

The credential data structure

The definition of this credential data structure must be exactly the same as the definition in the ACME agent. Any differences between the data structure definitions cause problems, including a difference as small as a buffer size.

Persona Extension Block (PXB)

PXB is the data structure to store the persona extension data. The header fields are pre-defined and must always be defined in a PXB. If the PXB is defined as a struct pxb_p1 as in the example code, its header fields are defined as follows:

- struct pxb_p1 *pxb_p1$l_flink;
- struct pxb_p1 *pxb_p1$l_blink;
- unsigned short int pxb_p1$w_size;
- unsigned char pxb_p1$b_type;
- unsigned char pxb_p1$b_subtype;

The remaining fields in the PXB are the credentials. All fields in the credential data structure should be appended after the header part.

For more information about PXB, refer to Section C.3 in the *ACME Developer's Guide*.

Implement persona extension routines

Initialization routine (mandatory)

int persona_ext_initialize ();

The role of this routine is to declare the addresses of other person extension routines. All routines in the persona extension image are registered during a system boot. NSA$REGISTER_PSB_EXTENSION() in this routine performs this registration. Two arguments are passed to this initialization function. The first argument is a descriptor containing a name of the extension image. The second argument is a Persona Extension Dispatch Vector (PXDV). Addresses of persona extension routines must be set to their corresponding fields (PXDV$A_CREATE, PXDV$A_CLONE, PXDV$A_DELEGATE, PXDV$A_DELETE, PXDV$A_MODIFY, PXDV$A_QUERY, and PXDV$A_MAKE_TLV) in PXDV. The details about NSA$REGISTER_PSB_EXTENSION() are available in Section 12.5 in the *ACME Developer's Guide*.

Assuming that the persona extension routines are named as persona_ext_create() and so forth, and pxdv is the PXDV declared in this routine, we can register the routines as follows in this initialize routine.

```
pxdv.pxdv$a_create   = (void *) persona_ext_create;      /* required */
pxdv.pxdv$a_clone    = (void *) persona_ext_clone;       /* optional */
pxdv.pxdv$a_delegate = (void *) persona_ext_delegate;    /* optional */
pxdv.pxdv$a_delete   = (void *) persona_ext_delete;      /* required */
pxdv.pxdv$a_modify   = (void *) persona_ext_modify;      /* required */
pxdv.pxdv$a_query    = (void *) persona_ext_query;       /* required */
pxdv.pxdv$a_make_tlv = (void *) persona_ext_make_tlv;    /* required */

status = nsa$register_psb_extension(&p1_desc, &p1_pxdv);
```

Create routine (mandatory)

```
int persona_ext_create (
    PSB *psb,
      PXB **pxb,
      P1_CREDENT *credential,
      unsigned int credential_size
    );
```

This routine creates a new persona extension. Specifically, this routine allocates a PXB in the non-paged pool and sets credential values in the PXB. For PXB allocation in the non-paged pool, EXE_STD$ALONONPAGED() is used. Credentials are copied into the credential data structure defined in this program.

Clone routine (optional)

```
int persona_ext_clone (
        PSB *psb,
        PXB *pxb,
        PXB *new_pxb
        );
```

This routine copies an existing persona extension in the context of the current process. An OpenVMS application can request this operation by calling the $PERSONA_CLONE system service. In the Clone routine, another extension-specific PXB is allocated with EXE_STD$ALONONPAGED(), and the system PXB is copied to the newly allocated PXB. Before returning SS$_NORMAL, the address of the new PXB must be passed to the new PXB provided as the third argument of this routine. The implementation of this routine is optional. If it is not implemented, no persona extension is created in the new persona during the $PERSONA_CLONE operation.

Delegate routine (optional)

```
int persona_ext_delegate (
        PSB *psb,
        PXB *pxb,
        int unused,
        PXB *new_pxb,
        PSB *new_psb
        );
```

This routine copies an existing persona extension into a different process. This routine is invoked by the $PERSONA_DELEGATE system service in an application program. The implementation is similar to the clone routine. A new extension-specific PXB is allocated with EXE_STD$ALONGPAGED(). After the original PXB is copied to the new PXB data structure, the address of the new PXB is given to the routine’s forth argument.

Delete routine (mandatory)

```
int persona_ext_delete (
        PSB *psb,
        PXB *pxb
      );
```

In this routine, the PXB data structure is deleted with EXE_STD$DEANONPAGED(). In general, the implementation of this routine is simple. Unless the PSB or PBX is empty, the whole PXB is deallocated with EXE_STD_DEANONPAGED(). But the way of deletion depends on the clone and delegate implementation.

Modify routine (mandatory)

```
int persona_ext_modify (
        PSB *psb,
        PXB *pxb,
        int itemcode,
        char *buf_addr,
        int buf_len
        );
```

This routine modifies data in the persona extension when the application calls the $PERSONA_MODIFY system service. In this routine, the item code provided as a third argument is modified to the value stored in the buffer in the forth argument. If the input item code is not supported, SS$_BADITMCOD must be returned. If the modification operation is successful, SS$_NORMAL will be the return value.

Query routine (mandatory)

```
int persona_ext_query (
        PSB *psb,
        PXB *pxb,
        int itemcode,
        char *buf_addr,
        int buf_len,
        int *ret_len,
        int queryflg,
        struct dsc$descriptor_s *dsc
        );
```

This routine retrieves a credential field requested by the application calling the $PERSONA_QUERY system service. The following item codes must be supported:

- ISS$_COMMON_FLAGS
- ISS$_DOI
- ISS$_COMMON_USERNAME
- ISS$_DOMAIN
- ISS$_COMMON_PRINCIPAL
- ISS$_COMMON_ACCOUNT
- ISS$_EXTENSION

If the input item code is not supported in this routine, SS$_BADITMCOD will be returned.

When queryflag is turned on, this routine compares the input data to the one in PXB. For every item code, the flag is checked first, and then both “compare” and “retrieve” modes should be implemented. The following lines are an example.

```
case ISS$_DOI:
    if (queryflg == COMPARE) {
                /* compare buffer */
                if (strcmp(buf_addr,  pxb_p1_p->pxb_p1$domain) != 0)
                        status = FALSE;
                else
                        status = TRUE;
    }else{
                if (buf_len >= sizeof(pxb_p1_p->pxb_p1$domain)){
                        strncpy(buf_addr, pxb_p1_p->pxb_p1$doi,
                                    sizeof(pxb_p1_p->pxb_p1$doi));
                        ret_len = sizeof(pxb_p1_p->pxb_p1$doi);
                }else
                        status = SS$_BADBUFLEN;
    }
    break;
```

Make_TLV routine (mandatory)

```
int persona_ext_make_tlv (
    PSB *psb,
    PXB *pxb,
    int itemcode,
    char *buf_addr,
    int buf_len,
    int *ret_len,
    int flags
    );
```

This routine is available with the intention to package credentials into a position-independent string for batch jobs. However, implementation of this routine is rarely required at this point. Thus, it is sufficient to simply return SS$_UNSUPPORTED in most cases.

Configure ACME — Put all the components together

Now we have all of the pieces for authentication with the ACME subsystem. For ACME agents to work properly, careful configuration of the ACME components, which are explained in the previous sections, is necessary. Follow the steps below.

Installing the persona extension image

The installation of a persona extension requires more than just copying the image into SYS$LOADABLE_IMAGES. The Alpha image should be tested with CHECK_SECTIONS.COM, a utility to check that the executive image is loadable. In addition, run a SYSMAN command to load the persona extension image, and then execute VMS$SYSTEM_IMAGES.COM to generate a new system image data file. After all this is done, reboot the system. The DCL and SYSMAN commands for these operations can be found in Appendix B.

Configuring the SYSUAF flags and security policy bits

If your ACME agent performs authentication, you must set either the EXTAUTH flag in the SYSUAF record for each VMS account to use external authentication or the IGNORE_EXTAUTH security policy bit. If either flag is not set in the user account and you enable the VMS agent first, the VMS agent handles authentication requests.

For an untargeted $ACM call, an ACME agent including the VMS agent becomes the Designated DOI agent if it declares DOI through the ACMEKCV$CB_SET_DESIGNATED_DOI() callback function.

When an ACME agent (other than the VMS agent) becomes the Designated DOI agent and performs authentication, the VMS agent synchronizes the password with the mapped user account in the SYSUAF file. For example, after a user, Mike, is authenticated by ACME agent X in which Mike's password is "password_x," Mike's password in SYSUAF will be updated to password_x. To disable this automatic password synchronization, the DISPWDSYNCH flag can be set in Mike's account in SYSUAF.

To enforce the same effects as the EXTAUTH and DISPWDSYNCH flags for all user accounts in the entire system, the IGNORE_EXTAUTH and GUARDS_PASSWORD bits in the SECURITY_POLICY system parameter bitmask can be used. The SECURITY_POLICY bit can be modified by the SET SECURITY command with the SYSGEN utility. After changing a value in the SECURITY_POLICY bit, the system must be rebooted to enforce the new value.

For more information about those flags and bits, refer to Section 1.10 in the *ACME Developer's Guide*. Values of the SECURITY Policy bits can be found in Chapter 7 in the *OpenVMS Guide to System Security*.

Configuring the ACME subsystem

The ACME agent images must be copied to SYS$LIBRARY, and then the ACME subsystem can be configured with SET SERVER ACME commands shown below. There are several states of the ACME subsystem, and the SHOW SERVER ACME command displays the current state.

Figure 5 shows the ACME subsystem's state transitions with SET SERVER ACME commands.

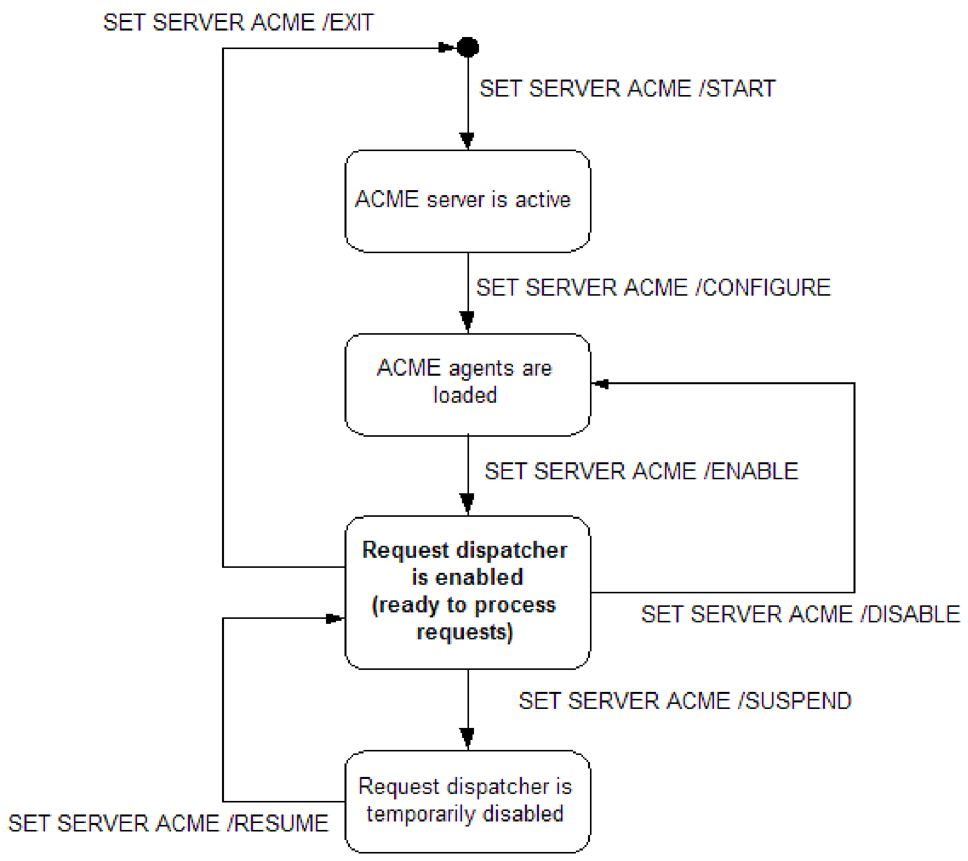


Figure 5: SET SERVER ACME commands and states of the ACME subsystem

$ SET SERVER ACME/START

This command is the first step—it starts the ACME server.

$ SET SERVER ACME/CONFIGURE=(NAME=*agent_name*, CREDENTIAL=*credential_name*)

After the ACME server gets started, this command can be run to load an ACME agent.

To load the VMS ACME agent:

$ SET SERVER ACME/CONFIGURE=(NAME=VMS, CREDENTIAL=VMS)

To load the example ACME agent (ACME_EXAMPLE_DOI) with the example persona extension (P1):

$ SET SERVER ACME/CONFIGURE=(NAME=ACME_EXAMPLE_DOI, CREDENTIAL=P1)

```
$ SET SERVER ACME/ENABLE
```

This command activates ACME agents that have already been loaded in the ACME subsystem. The order of ACME agents is specified by this command. The following commands demonstrate the order of the VMS and example ACME agents.

The VMS agent is the first agent.

```
$ SET SERVER ACME/ENABLE=NAME=(VMS, ACME_EXAMPLE_DOI)
```

The example agent is the first agent.

```
$ SET SERVER ACME/ENABLE=NAME=(ACME_EXAMPLE_DOI, VMS)
```

```
$ SET SERVER ACME/DISABLE
```

This command disables the ACME server and agents.

```
$ SET SERVER ACME/EXIT
```

Run this command to stop the ACME server. Once this command is executed, all of the ACME agents are unloaded because the ACME subsystem stops.

Perform a request from the $ACM application

Once the ACME subsystem has been configured, it is essential to send requests from the $ACM application that will be used in the production environment. If the ACME agent issues credentials, the $ACM application must be capable of handling them. Ensure that the proper $ACM application is used for testing. The ACMEUTIL example program is provided to test the example ACME agent, which issues a simple username and principal as credentials,. Note that Telnet and ftp can be used to send authentication requests, but they are not directly calling the $ACM system service. The authentication with Telnet, ftp, and SET HOST is invoked by the $ACM service in the ACME LOGINOUT image.

Sending an untargeted request

```
$ ACMEUTIL AUTH /PERSONA/DIALOGUE=(INPUT,NOECHO)
```

Sending a request targeting the ACME_EXAMPLE_DOI agent

```
$ ACMEUTIL AUTH /PERSONA /DIALOGUE=(INPUT,NOECHO) /DOMAIN=
ACME_EXAMPLE_DOI
```

Sending a request targeting the VMS agent

```
$ ACMEUTIL AUTH /PERSONA/DIALOGUE=(INPUT,NOECHO)/DOMAIN=VMS
```

Summary

The ACME subsystem provides a new authentication environment on OpenVMS. To enforce new authentication policies, we can load ACME agents in a "plug-in" manner. Many OpenVMS system managers will benefit from the flexibility of this new capability. As OpenVMS developers, we can create new ACME agents for new authentication policies.

The major task for developing an ACME agent is to implement every callout routine for its authentication policy. By referring to the steps in every callout routine in the Implement an ACME Agent section as well as the example ACME agent's source, ACME agent developers will have clearer ideas about how to implement ACME agents.

If the ACME agent issues credentials, it is also necessary to develop its persona extension. A persona extension is an executive image that securely stores credentials for the $ACM application process/thread. The Implement a Persona Extension section provides comprehensive steps for developing a persona extension. It is recommended that you read this part with the source code of the sample persona extension.

Finally, after the ACME agent and its persona extension become available, ACME developers and OpenVMS system managers have to know how to install and configure all the components for testing and setting up production environments. Comprehensive steps for installing and configuring an ACME agent and persona extension are available in the Configure ACME section.

For more information

- *ACME Developer's Guide* (SYS$HELP:ACME_DEV_GUIDE.PDF)
- Chapter 33. Authentication and Credential Management (ACM) System Service, *OpenVMS Programming Concept Manual* (http://h71000.www7.hp.com/doc/731FINAL/5841/5841pro_contents_010.html#toc_chapter_33)
- *OpenVMS Guide to System Security* (http://h71000.www7.hp.com/doc/732FINAL/aa-q2hlg-te/aa-q2hlg-te.PDF)
- *HP OpenVMS System Services Reference Manual*: GETUTC–Z (http://h71000.www7.hp.com/doc/732FINAL/DOCUMENTATION/PDF/aa-qsbnf-te.PDF)

Acknowledgements

While I was writing this article, I received tremendous support from many people in the OpenVMS Engineering Group. I spent lots of time with the members of the ACME project, Rick Barry, John Harney, Doug Fyfe, and Barbara Thomson, about the details of the ACME and persona extension architectures. The feedback from Rick, Barbara and John for the drafts was invaluable. I also thank the editor of this article, Kathleen Johnson, for her extensive review of this article.

Appendix A: How to Build and Set Up the Example ACME Agent (ACME_EXAMPLE_DOI_ACME.C)

1. Compile the message file (if it hasn't been done)

```
$ MESSAGE ACME_EXAMPLE_DOI_MSG.MSG
```

   - This command creates ACME_EXAMPLE_DOI_MSG.OBJ, which will be linked later

2. Build (compile and link) the example ACME agent
   (This command creates VMS$ACME_EXAMPLE_DOI_ACMESHR.EXE)

```
$ @ACME_EXAMPLE_DOI_BUILD.COM
```

3. Copy the example ACME agent image to SYS$LIBRARY

```
$ COPY VMS$ACME_EXAMPLE_DOI_ACMESHR.EXE SYS$LIBRARY
```

Appendix B: How to Build and Set Up the Persona Extension Example (ACME_PERSONA_EXT.C)

1. Build (compile and link) the example persona extension image
   (This command creates P1_EXT.EXE)

```
$ @ACME_PERSONA_BUILD.COM
```

2. Test the example persona extension image with SYS$ETC:CHECK_SECTIONS.COM (on OpenVMS Alpha only)

```
$ @SYS$ETC:CHECK_SECTIONS.COM P1_EXT.EXE
```

3. Copy the example persona extension image to SYS$LOADABLE_IMAGES

```
$ COPY P1_EXT.EXE SYS$LOADABLE_IMAGES
```

4. Install the example persona extension image
   (the image is P1_EXT.EXE, and the product name is ACMETEST)

```
$ MCR SYSMAN
SYSMAN> SYS_LOADABLE ADD/LOG ACMETEST P1_EXT

$ @SYS$UPDATE:VMS$SYSTEM_IMAGES.COM
```

5. Reboot the system

```
$ @SYS$SYSTEM:SHUTDOWN
```

During reboot, an error message appears if the persona extension Image is not loaded. If you don't see the error message, the image should be loaded properly.

To verify:

```
$ ANALYZE /SYSTEM
SDA> SHOW EXECUTIVE P1_EXT
```

Appendix C: How to Set Up the ACME Agent and Persona Extension After Reboot

1. Start the ACME server

```
$ SET SERVER ACME/START/LOG
```

2. Load the VMS ACME agent (this agent must be always loaded)

```
$ SET SERVER ACME/CONFIGURE=(NAME=VMS,CREDENTIAL=VMS)
```

3. Load the example agent with the example persona extension (P1)

```
$ SET SERVER ACME/CONFIGURE=(NAME=ACME_EXAMPLE_DOI,CREDENTIAL=P1)
```

4. Enable the agents (the order is the example agent is first)

```
$ SET SERVER ACME/ENABLE=NAME=(ACME_EXAMPLE_DOI,VMS)
```

Appendix D: How to Test the Example Agent from the ACMEUTIL Client Program (in SYS$EXAMPLES)

ACMEUTIL is a DCL utility program executing the $ACM[W] system service for authentication and change-password requests.

Authenticate Principal request (targeted call, credential is requested)

```
$ acmeutil auth /persona/dialogue=(input,noecho)/domain=acme_example_doi
```

Change Password request (Untargeted call)

```
$ acmeutil change dialogue=(input,noecho)
```

For more information about this program, see ACMEUTIL_SETUP.COM in SYS$EXAMPLES.

Appendix E: How to Install the ACME LOGINOUT Image

The ACMELOGIN kit is provided to install versions of LOGINOUT.EXE and SETP0.EXE that are modified to use the SYS$ACM system service. Since these images use SYS$ACM, they will use the authentication policies provided by the ACME agents that have been configured on your system including user-defined agents.

Note: It is recommended that you first test your ACME agent using the ACMEUTIL utility described earlier in this document before installing the ACMELOGIN kit.

Three PCSI kits are contained in the BACKUP saveset SYS$UPDATE:ACME_DEV_KITS.BCK. Restore the PCSI kits to your default directory using BACKUP:

```
$ BACKUP SYS$UPDATE:ACME_DEV_KITS.BCK/SAVE *.*
```

This will create three PCSI kits:

```
DEC-AXPVMS-V732_ACMELOGIN-V0100--4.PCSI        (ACMELOGIN V1.0 patch kit)
```

The ACMELOGIN kit contains modified versions of LOGINOUT.EXE and SETP0.EXE that use the SYS$ACM system service to perform authentication and password changes.

```
DEC-AXPVMS-V732_LOGIN-V0100--4.PCSI       (LOGIN V1.0 patch kit)
```

The LOGIN kit contains the original LOGINOUT.EXE and SETP0.EXE images that were shipped with this release. You can install this kit to restore the original versions of these files if you've previously installed the ACMELOGIN kit for development and testing.

Note: If you previously installed any ECO kits that modified LOGINOUT.EXE or SETP0.EXE, you will need to re-apply those ECO kits after restoring the original images using the LOGIN kit.

```
DEC-AXPVMS-V732_ACMELDAP-V0100--4.PCSI  (ACMELDAP V1.0 patch kit)
```

The ACMELDAP kit contains the LDAP ACME sharable image, management tools, a startup file, CLD file, and initialization template, as well as the .PS, .TXT and .HTML documentation.

Install the kit using the Polycenter Software Installation Utility from a privileged account.

To install the ACME LOGINOUT image:

```
  $ PRODUCT INSTALL V732_ACMELOGIN
```

To install the traditional LOGINOUT image:

```
  $ PRODUCT INSTALL V732_LOGIN
```

Appendix F: ACME Terminology

Agent-specific item codes

The set of extended item codes that are defined by an agent and known only to that agent and any customized $ACM applications. An agent never prompts a generic $ACM application for agent-specific item codes unless the item code represents a simple text-based data element that a generic $ACM application can process blindly. For example, an agent can prompt a generic $ACM application for an agent-specific item code representing a token id string which can be responded to by a human user, but the agent is not allowed to prompt a generic $ACM application for specialized, binary data such as might be used in a hardware token.

Auxiliary agent

An agent that implements a partial authentication policy or some function such as password filtering, but cannot issue credentials. It cannot be the target of an $ACM call. An auxiliary agent logically works in conjunction with the designated DOI agent.

Common item codes

The set of basic item codes documented by the $ACM system service that exists for every OpenVMS system and is recognized by all agents and $ACM applications. All common item codes can be specified on the initial $ACM call, but only a subset of well known common item codes may be processed in dialogue mode (see below). Examples of common item codes are:

ACME$_LOGON_TYPE

ACME$_AUTH_MECHANISM

ACME$_NEW_PASSWORD_FLAGS

ACME$_PRINCIPAL_NAME_IN

ACME$_PASSWORD_1

Cooperative model

For untargeted $ACM calls, an DOI agent in the cooperative model enforces authentication and issue credentials using a single principal-name and password scheme as seen from the perspective of the $ACM application.

Credential

Information containing the user's identity, privileges, and roles within a given security environment.

Designated DOI agent

For targeted $ACM calls, the Designated DOI agent is the DOI agent specified in the $ACM call. For untargeted $ACM calls, the Designated DOI agent is generally the first DOI agent in the order of execution that locates the principal-name in its principal-name database. This is the only

DOI agent allowed to prompt for passwords. It always issues credentials if the authentication is successful and is responsible for the ultimate success or failure of the request unless the VMS agent cannot map the principal name.

Dialogue mode

The mode in which an agent issues a request to acquire information from the user (or to be displayed to the user). The calling application obtains the information from the user (or displays it to the user) and calls $ACM again to proceed until the service indicates that no further interaction is required. $ACM applications that specify the context argument operate in dialogue mode.

DOI

Domain-of-Interpretation. A DOI represents a security environment having a principalnamespace, authentication and authorization schemes, and information representing a user's identity (both VMS and DOI-specific) and privileges. A DOI agent implements a particular DOI. A DOI agent can be the target of an $ACM call.

Independent model

In the independent model, a DOI agent performs authentication and issues credentials only when it is operating as the designated DOI agent, otherwise it does not participate in the request.

LOGINOUT

Two different LOGINOUT images are shipped with the OpenVMS Alpha operating system. To use the ACME subsystem, the ACME LOGINTOUT image must be installed. The other image, LOGIN82, is used for the traditional $LGI authentication. The procedures to install ACME LOGINOUT will be described in Appendix E.

Phase

A phase is a discrete stage of request processing. Each phase is associated with a callout routine within an agent that the ACME server invokes.

Principal-Name

A string representing a user (sometimes referred to as username).

Request

An $ACM request is represented internally as a *work queue entry* (WQE). The WQE is used to maintain the state of the request through multiple stages of processing. It is also used to control certain interactions among the agents.

Secondary DOI agent

A DOI agent is one that is not operating as the designated DOI agent. It may perform authentication and issue credentials.

Targeted call

A call to $ACM that specifies the ACME$_TARGET_DOI_ID or ACME$_TARGET_DOI_

NAME item code.

Untargeted call

A call to $ACM that does not specify the ACME$_TARGET_DOI_ID or ACME$_

TARGET_DOI_NAME item code.

Well-known item codes

The set of common item codes that an $ACM application can expect to process in dialogue mode (or to supply in a single non-dialogue $ACM call). Generic $ACM applications can respond to well-known item codes, even in restricted operating environments where there is no human user with which to interact or where application protocols accept only username and password data. Examples of well-known item codes are:

ACME$_PASSWORD_SYSTEM

ACME$_PRINCIPAL_NAME_IN

ACME$_PASSWORD_1